INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

C++ Classes for 2-D Unstructured Mesh Programming

R. Bruce SIMPSON

No 3592

December 23, 1998

THÈME 4

ISSN 0249-6399

C++ Classes for 2-D Unstructured Mesh Programming

R. Bruce SIMPSON *

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Gamma

Rapport de recherche n ° 3592 — December 23, 1998 — 55 pages

Abstract: In this report, a set of `C++` classes is presented for representing unstructured triangular meshes of intrinsic dimension two; i.e. oriented 2-manifolds. For simple mesh objects, i.e. vertices, triangles, and line segments, a small set of class members that are sufficient for the mesh class are described. They define abstractions based on their incidence relations and a few geometric primitives for a mesh class, which is an intelligent container class of three lists of these simple mesh objects.

The classes are intended to be components in an object oriented approach to software for meshing applications described in the report. This context differentiates the roles of the mesh class and the simple mesh object classes; these latter can be extended as the carriers of the applications data. The capability of the classes of this report to simultaneously simplify the coding of mesh methods and facilitate generalization of the code is discussed with examples. The report provides an overview of the class design and use, tutorial examples, and, in a large appendix, class documentation details.

Key-words: unstructured mesh programming, object oriented, C++

(Résumé : tsvp)

* Visiting from the Department of Computer Science, University of Waterloo, Canada. The hospitality of the Gamma project during the preparation of this report is gratefully acknowledged. This research has also been supported by the Natural Science and Engineering Research Council of Canada (`http://www.nserc.ca/`) and Communications and Information Technology Ontario (`http:/www.cito.ca/`) **E-mail: rbsimpson@uwaterloo.ca**

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Classes en C++ pour programmation des methodes des maillage surfacique

Résumé : Ce rapport présente un ensemble de classes C++ permettant la représentation de maillages triangulaires non-structurés intrinsèquement de dimension deux, *i.e.,* des surfaces orientées.

On indique quels sont les attributs de ces classes nécessaires à la description des objets simples rencontrés, les sommets, les triangles et les segments. Ils déterminent les relations liées aux liens topologiques présents et les quelques primitives de nature géométrique de la classe maillage.

Les classes sont définies pour former les constituants d'une approche orientée objet utile au développement et à l'implémentation de logiciels de maillages. Le contexte fait la différence entre les roles de la classe maillage et les classes des objets simples intervenant. Ces dernières sont modifiables afin de pouvoir prendre en compte d'autres données jugées nécessaires pour applications envisagées.

Le but des classes introduites est de simplifier le codage des méthodes de construction de maillages et de permettre leur généralisation ou extension comme on le montre sur quelques exemples.

Ce rapport présente donc une description des classes introduites. Il indique comment les utiliser et donne des exemples simples. Par ailleurs, une annexe précise les aspects techniques des classes discutées.

Mots-clé : maillage programmation, approche orientée objet, C++

Contents

1 Introduction

We report on a set of C++ classes for representing unstructured triangular meshes of intrinsic dimension 2, i.e. oriented 2-manifold meshes. Unstructured triangular meshes are efficient, flexible 2 dimensional discretizations. However, the programming of methods that exploit these advantages is too complex for one-off efforts to be efficient from a program development viewpoint. Hence this programming effort can benefit from modularization using software components, and the classes described in this report are intended to play such a role. These classes support mesh generation in essentially arbitrary global geometries as we elaborate below. Three simple mesh object classes are introduced for mesh vertices, triangles, and line segments, and a mesh class which is a container class for three lists of simple mesh object classes. As is commonly pointed out, effective abstractions of simple mesh objects can be formed by separating topological data , i.e. incidences, from geometric detail, usually with the addition of a few geometric primitives to aid the abstraction. The simple mesh object class declarations define this separation, and the mesh class only 'knows' about the abstractions of the mesh objects. In this way, the mesh class is largely geometry independent. The implications for simple mesh object hierarchies are discussed in §4.

This report has several purposes. It is intended as a report on this and other research into class design for mesh generation and applications. It also serves as documentation for the classes presented and includes tutorial material on their use. A basic familiarity with C++ is assumed, but familiarity with object oriented programming (OOP) is not; the few relevant aspects of OOP involved are explicitly described. In §2 and §3, we give an overview of the classes with general descriptions of their data members and functionality. Details in terms of header files and function member specifications are provided in Appendix §A.1 and Appendix §A.2. In §4, we set these classes in the context of OOP design of mesh generation and applications software to differentiate the roles of the simple mesh object classes and the mesh class. A small tutorial example is described in §5; the details of the implementation are given in Appendix §A.3.

The classes are, in effect, software components for writing mesh generation code; e.g. refinement with Delaunay insertion or advancing front technique codes. The abstraction of the simple mesh objects used by the mesh class can also be used by the implementations of the mesh generation methods to produce code that is highly geometry independent, as we elaborate in §4 and §5. The result is mesh generation code that is reuseable in the sense that it can be linked to various application specific simple mesh object class implementations. This follows the concepts of OOP design

that have been presented for stiffness matrix generation in the finite element method(Mackie '92 [10], revue européen des Eléments finis '98 [2]) and which we discuss further in §4. In fact, it would be natural to construct mesh object classes that combine the geometry specific members of the simple mesh object class of this report with the physical modeling and shape function members of the OOP FEM classes.

A mesh representation, this one or other, plays a well known role in software design of defining a software programming interface for modularity and program components. The primary goals of this design are the usual OO objectives of simplifying programming while simultaneously facilitating generalizing the program parts; i.e. increasing code reuseability. This programming simplification is sought through intellectually manageable abstractions and applies not only to code creation initially, but perhaps even more importantly, to subsequent code maintenance, where much code reuseability is actually realized.

In an application area for which these techniques are relatively new, such as meshing, it is not likely that one can *a priori* anticipate the extent to which specific representation classes can reach these goals. It seems necessary to build some prototypes and gain some direct experience. Moreover, the merit of such representations does not rest solely on these goals. A representation that offers benefits in code development/maintenance efficiencies must also be reasonably competitive in run time efficiency. This work is still in a state of providing direct experience with the representation; we have been using versions of these classes for about four years. Although efficient implementation has not been a first priority during this time; several considerations have influenced the design in anticipation of this need. Our choice of abstractions has been guided by the need to be able to implement any of the more efficient algorithms associated with meshing applications. The mesh class has been implemented so that it does not require inheritance in the simple mesh object classes, (neither does it preclude it.) Inheritance would provide helpful approaches to several implementation issues as we discuss in §4. But our initial experience confirms that of others; i.e. run time linking associated with inheritance for small objects can impose a high execution time cost. A closely related issue is the separate compilability of the implementations of the mesh class and the simple mesh object classes; this issue is also discussed briefly in §4.

Many of the ideas in this report have close connections with other meshing software designs. Although we make some references to these throughout the text, we primarily reserve comparative comments on related work to §6. Although the OOP approach is actively reported in the engineering literature on the FEM, details of class design are rare. More is available in closely related domain of com-

putational geometry programming, however. The library under development in the CGAL project is well documented in the report by Fabri et al , 1998 [5], and on-line at `http://www.cs.uu.nl/CGAL/`; the report references much related computational geometry software. The class designs of this report and their roles in mesh programming are connected to the 2 dimensional triangulation classes of the CGAL project. More directly connected to meshing applications but less explicit, are the codes described in Mobley, Carroll, and Cannan, [11], Palmer [13] , and Vavasis; see §6 for further comments.

2 Geometry of the class objects

In this section, we will add to the familiar geometry of 2-D meshing some precision about the terminology and properties of the objects used in connection with the **Mesh2D** classes. We will restrict the discussion to features as used to mesh oriented polyhedral surfaces to reduce the technicalities of the description, although the classes are designed to be used for general oriented surfaces.

Geometry of the simple mesh objects and meshes

Directed *edges* join *vertices* in 2 or 3 space and are directed from an origin vertex to a destination vertex, i.e. they are ordered pairs of vertices.

Triangles are cycles of three directed edges which are locally numbered 0,1,2. Of course, the destination of each edge is the origin of the next edge. In meshes, these cycles must have a counter clockwise direction described further below. Triangles t and s are neighbours if there is an edge e which belongs to t and such that -e belongs to s.

Line segments are bidirectional edges, or pairs of edges that join the same two vertices but with opposing directions. The two directions are locally numbered 0 and 1. This view allows the neighbouring relation to be extended to a triangle t and a line segment s, so as to permit a line segment to have two neighbouring triangles.

In a mesh, we expect the triangles to fit together to tile some two dimensional region i.e. to abut each other as neighbours. Most of the edges between neighbours are artifacts of the mesh; however, applications typically specify some edges as either boundaries or internal interfaces. Line segments are introduced to distinquish these edges, but to do so in a relatively inobtrusive way from a programming point of view, i.e. by being basically special kinds of triangles. In FEM applications, line segments may be finite elements themselves, carry physical modeling and shape function data. Figure 1 shows an exploded view of some technicalities of the neighbours relation. Triangles $\mathtt{t}_1$ and $\mathtt{t}_2$ are neighbours on their edges locally numbered 0 and 1 respectively. Line segment `s` is shown as a pair of directed edges between vertices Q and R; the edge numbered 0 is directed from Q to R. Triangle $\mathtt{t}_1$ and line segment `s` are neighbours on their edges numbered 2 and 1 respectively and triangle $\mathtt{t}_3$ and `s` are neighbours on their edges numbered 0 for both. However, we do not regard $\mathtt{t}_1$ and $\mathtt{t}_3$ to be neighbours, in consequence of the presence of `s`; in fact, this is just the point of introducing `s`. Every triangle then can be viewed as having three neigh-

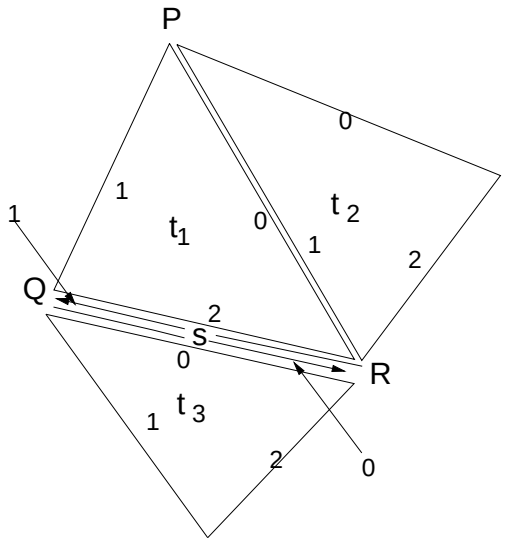


Figure 1: A configuration of neighbours

bours, which may be other triangles, or line segments, or `NULL` - as on edge 1 of $\mathtt{t}_1$. Similarly, every line segment has two neighbours, but these are restricted to being either triangles or `NULL` , not other line segments.

Meshes are a triple of sets: a set of vertices, a set of triangles, and a set of line segments. We do not allow isolated vertices, but otherwise one of the other two sets can be empty; e.g. a mesh with only line segments could be the boundary representation of domain to be meshed, or a planar straight line graph (PSLG) (see Bern & Eppstein, 1992 [1]) to be triangulated.

Each mesh in 3-D is associated with a positive direction vector pointing into the half space on its positive side. Meshes do not have to be strictly planar, but every triangle of a mesh and its neighbours, must be visible from sufficiently far out on a ray emerging from the mesh in its positive direction. The local numbering of the cycle of edges of a triangle in the mesh should result in a counter clockwise cycle when seen from the positive side of the mesh plane. Two dimensional meshes can be considered to lie in the (x, y) plane of 3-D and obey this convention with the positive z axis as the normal.

Vertices, triangles, and line segments each carry a positive global index, (also referred to as its label) which must be unique in a given mesh. These indices do not

need to be consecutive, i.e. a contiguous range of integers.

Application domains often have natural decompositions for which it is convenient both logically and for programming to assign individual meshes [1] , see multiblock meshing techniques Chapter 9, P. L. George [6]. For example, it may be convenient for meshing polyhedral surfaces to identify a mesh with each face. The mesh class supports programming with collections of meshes abutting on common boundaries marked by line segments for these purposes. We refer to such collections of related meshes as a *composite* mesh. It is common in the literature to regard this global mesh as *the* mesh and its components as submeshes, but for our class definitions it is more convenient to use this composite mesh terminology.

Figure 2 shows a very small 2-D composite mesh made up of two meshes A and B each with 2 triangles, and with line segment labels set in square boxes. Figure 3 shows a exploded view of a boundary representation of a cube with a corner removed which is a composite mesh of 9 faces bounded by line segments and without triangles.

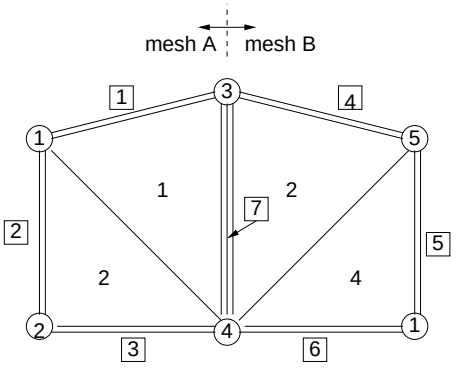


Figure 2: a small composite mesh

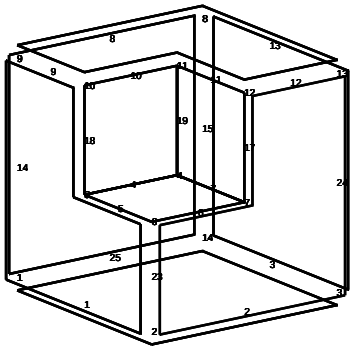


Figure 3: exploded cut out cube

The incidence relations between simple mesh objects and meshs of a composite mesh are:

- a triangle can belong to only one mesh
- a line segment can belong to two meshes, if it is part of an internal boundary of a composite mesh
- a vertex can belong to many meshes

[1] also called surface patches

As Figure 2 shows, the label for each mesh component is to be positive, and unique in each mesh that it is registered in. It is not necessarily unique in the composite mesh, nor are these labels necessarily consecutive. In particular, a globally unique numbering of the simple geometric objects in a composite mesh induces an acceptable labeling for each (sub)mesh.

3 Introducing the classes

We now introduce the main features of the `C++` data types for the geometric objects just described, with a few details needed for the example of §3.2. The mesh representation data declared in the simple mesh object classes for vertices, triangles and line segments are essentially standard in finite element mesh programming. In an object oriented approach to programming mesh based applications, the classes for these mesh objects may also contain applications and geometric data in addition to that needed for mesh representation alone, as we discuss further in §4 and §5. As seen by the mesh class, the mesh object classes define abstractions of their geometric counterparts that primarily carry local incidence data. As seen from the mesh class, their functionality is limited to accessing it. The capability to change this data is encapsulated in four powerful editing function members of the **Mesh2D** class discussed in §3.1. First we present the simple mesh object classes as seen by the mesh class.

To present a quick conceptual overview of the definitions of incidence data members in these classes, we use some entity-relationship notation borrowed from data modeling (Simpson, 1997 [14]). In Figures 4 to 8, `C++` class names are in larger bold type and model entities which do not have `C++` identifiers are in smaller, italic type. More details can be found in the header files of Appendix §A.2.

MVert is the class for a vertex . Figure 4 shows it as a relation , or a tuple, involving global label **GVnum**, *coordinates* which are some unspecified form of coordinates, and an adjacency list of **MTri** or **MLSeg** which are incident on the vertex. *edge_*__**MTri**__, shown in Figure 5 characterizes the data associated with a directed edge of a triangle, i.e. each edge is associated with a local edge number,*edgeNo*, an **MVert** , which is opposite the edge, and either a **MTri** or a **MLSeg**, which is the abutting neighbour on this edge. The view of mesh triangles here is edge oriented; i.e. the *edgeNo* is used to access triangle vertices or neighbours in coding using these classes.

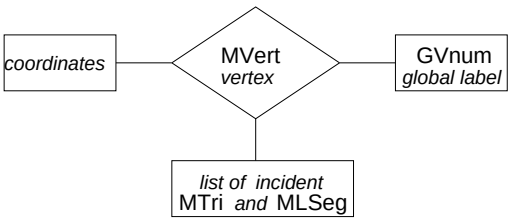


Figure 4: the **MVert** relation

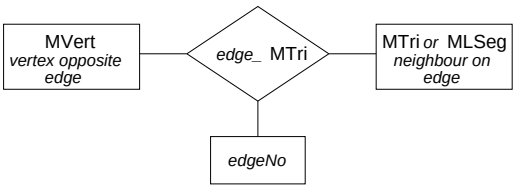


Figure 5: the edge_**MTri** relation

MTri is the class for a triangle . Figure 6 shows it as a 4 item tuple with an integer label **GTnum** and three *edge*_**MTri**s. It indicates that local addressing of triangle data is based on edge numbering. Hence, edge i has for its origin (destination) the vertex opposite edge $(i+1)\ mod\ 3$ (edge $(i+2)\ mod\ 3$).

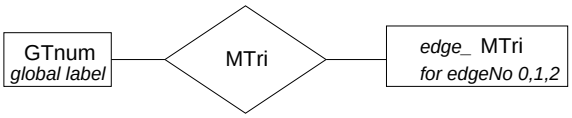


Figure 6: the **MTri** relation

*edge*_**MLSeg** is a relation characterizing the data associated with an edge of a line segment, as shown in Figure 7. It is similar to the *edge*_**MTri** relation with the addition of a **Mesh2D** datum which indicates the mesh to which the neighbouring **MTri** datum belongs. Note that only **MTri** data are permitted to be neighbours. **MLSeg** is the class for a line segment . Figure 8 shows it as a 3 item tuple with an integer label **GLSnum** and two *edge*_**MLSeg**s. The two *edge*_**MLSeg**s correspond to the two directions of a single undirected line segment; they carry different data in the different directions [2].

The convention for connecting vertices to edges in a **MLSeg** is that the vertex opposite e is the destination vertex of e. The convention for edge based indexing

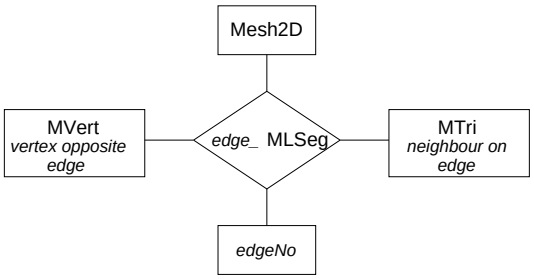


Figure 7: the *edge*_**MLSeg** relation

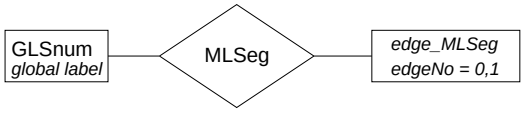


Figure 8: the **MLSeg** relation

of **MLSeg** data can be illustrated using Figure 2. Consider the **MLSeg** object of

[2] Since edges are actually directed edges and occurs in cycles they could be identified with the half edges used in representations for polyhedral surfaces, e.g. Kettner, 1998 [9]. Single (half) edges are not, however, explicitly represented in these classes as indicated above.

GLSnum = 7. If we set the edge locally numbered 0 to be the edge from the **MVert** of label 4 to that of label 3, then **MTri** of label 2 in mesh B will be the neighbour of this **MLSeg** on edge 0, and the **MVert** of label 3 will be the vertex opposite edge 0.

Mesh2D is the class for meshes . Its data consists primarily of three lists; a list *Vertices* of **MVert** objects, a list *Triangles* of **MTri** objects and a list *Boundary References* of **MLSeg** objects. To be explicit,these are lists of pointers to the simple mesh objects They are implemented as hash tables using the object labels as hashing keys. A composite mesh could be represented by an array of **Mesh2D** objects, e.g. the cut out cube of Figure (3) can be represented by an array declared as `Mesh2D * cface[9]` As we note in §5 however, while it may be convenient, it is not necessary to use a separate mesh for each planar surface.

In Figure 9, we show a schematic of the representation of the simple composite mesh of Figure 2 using these classes. Four **MTri** object have been allocated dynamic memory space, and 6 **MVert** and 7 **MLSeg** objects. Two **Mesh2D** objects, `meshA` and `meshB`, have been constructed and pointers to the various mesh elements have been entered in the **Mesh2D** lists. See that **MVert** 3 is registered in both meshes, as is **MLSeg** 7. The simple mesh objects carry their local incidence data. Mesh tasks involving local incidences such as element patch based error estimation can be carried out without reference to the **Mesh2D** objects. Indeed, Lawson's oriented walk search to locate the triangle in a convex composite mesh which contains a given point can be carried out using these local incidences without explicit reference to the **Mesh2D** meshes.

While the **Mesh2D** class supports meshes with the neighbours incidence data of the simple mesh objects as described, it doesn't require all of it. A **Mesh2D** object could contain a *Triangles* list of **MTri** with all `NULL` neighbouring data, which could be a useful simplication for some application. Similarly, a line segment can serve as a convenient connector between triangles in adjacent meshs of a composite mesh, but it is not required. If a program using these classes either does not need this connection or keeps its own records, it could dispense with the common line segment neighbour. (See detailed specifications of Appendix §A.1.)

Class member functions
With respect to their incidence data, the simple object classes, **MVert, MTri,** and **MLSeg** , are basically abstract data types that only provide data access. E.g. the **MTri** class has members functions `GetVOE(int)` and `GetNOEint` which take a local

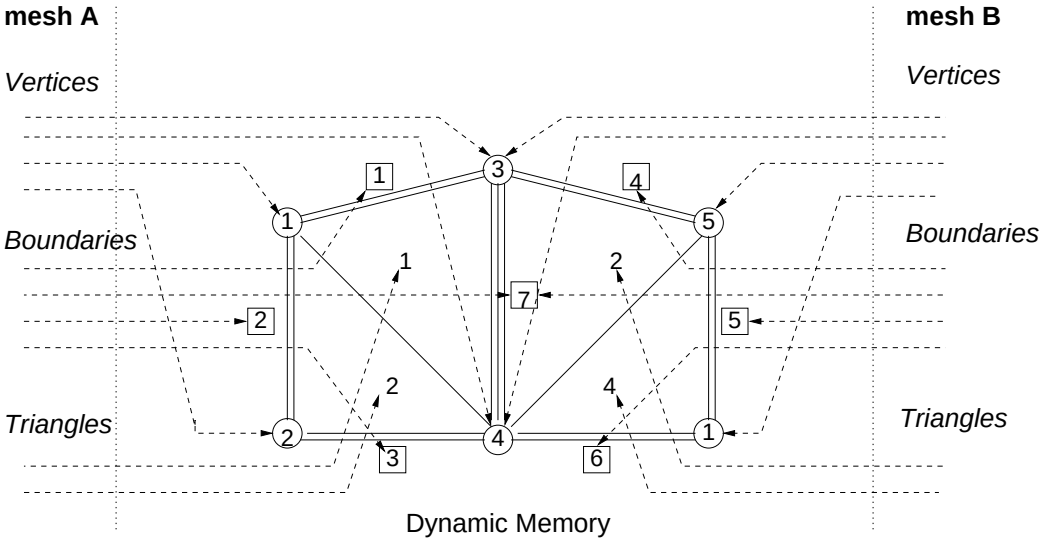


Figure 9: Schematic of Class objects

edge number as parameter and return a pointer to the vertex opposite this edge, or to the neighbour on this edge, respectively. It is common in mesh programming to require, for a shared edge, its local edge number in the neighbour; we will refer to this number as the complementary edge number. To provide this datum, the **MTri** class has a member function named `ComplEdgeNum(int)`, which has the property

$$\texttt{t} == (\texttt{t} \rightarrow \texttt{GetNOE(i)}) \rightarrow \texttt{GetNOE(t} \rightarrow \texttt{ComplEdgeNum(i))} \qquad (1)$$

The **MLSeg** class has been designed to be very similar to the **MTri** class to simplify the concept and programming of mesh neighbours. Indeed, conceptually one could think of **MLSeg** as a class derived from **MTri** . This point can be illustrated by the fact that relation (1) holds even if $t \rightarrow$ `GetNOE(i)` is a **MLSeg** object. However, we have not explicitly declared **MLSeg** as a derived class, nor overloaded the assignment operator. So, assigning a pointer to a **MLSeg** object to a **MTri** * variable requires an explicit type cast.

The simple mesh object classes also have a few validation member functions; a **MVert** can determine if its coordinates differ from those of another **MVert** ; a **MTri** can supply orientation information. However, essentially all of the incidence data editing capabilities are vested in **Mesh2D** member functions. For this purpose, the **Mesh2D** class has been declared a friend by the simple object classes. The **Mesh2D** class has two major categories of member functions: data *access* members and data *editing* members.

Mesh2D *data access members*

Access by label to pointers to the simple mesh objects registered in a **Mesh2D** object is provided by member functions named `XOfI(int)` for $X =$ `Vtx, Tri` or `LSeg`. Edges are identified by a pair involving a local edge number and a **MTri** or **MLSeg** pointer. For each list of simple mesh objects, the **Mesh2D** class provides a pair of member functions for scanning the list. For $X =$ `Vtx, Tri` or `LSeg`, `First`$X()$ returns a pointer to the first item in the list; subsequent calls to `Next`$X()$ return pointers to the subsequent items until `NULL` is returned. Some care is necessary if the list is modified during a scan, say by mesh refinement, to avoid generating infinite loops. The **Mesh2D** class also defines several list utilities, including a label server for each list. `NewG`X`num()` , for $X =$ `V, T` or `LS`, returns a label for a simple mesh object that is larger than any label ever registered in the relevant list (during the scope of the **Mesh2D** object).

3.1 Mesh2D Data Editing Functions

The data access function members just described are probably what the reader would anticipate from the preceding incidence data modeling discussion. But perhaps less obvious is how this design concentrates the data editing functions in four member functions of the **Mesh2D** class: `InsertTri`, `InsertLSeg`, `RemoveTri` and `RemoveLSeg`.

`Insert`Y, for $Y =$ `Tri` or `LSeg`, is the *only* way to:

- insert **M**Y or **MVert** pointers into a **Mesh2D**
- enter incidence data into a **M**Y
- add a **M**Y to the adjacency list of a **MVert**
- update the incidence data of a neighbouring **MTri** or **MLSeg** from `NULL`

`Remove` *Y*:
- removes an **M** *Y* pointer from a **Mesh2D**
- does not delete it from dynamic memory
- removes it from **MVert** adjacency lists
- sets neighbours incidences to `NULL`

The signature for `InsertTri` is

```
ErrCode InsertTri(MTri *t, MVert* vbuf[3], MTri* nbuf[3], int
                                cbuf[3]);
```

The signature for `InsertLSeg` is very similar; i.e.

```
ErrCode InsertLSeg(MLSeg *t, MVert* vbuf[3], MTri* nbuf[3], int
                    cbuf[3], Mesh2D * oppMesh);
```

Both of these member functions return the logical flag `ErrCode` with values `SUCCEEDED` for a successful insertion, `FAILED` otherwise.

The basic idea is to allow for a variety of possible ways to specify the data for the **MTri** to be inserted in *this* mesh. The **MVert** buffer array `vbuf` can specify up to three input vertices for **MTri** `*t` and, taken pairwise, `nbuf[i]`, `cbuf[i]` , `i = 0,1,2` form a buffer of up to three candidate edges for `*t`. The buffers may contain `NULL` or non-`NULL` data and should determine three vertices from which to construct **MTri** `*t`, which we refer to as the *candidate* vertices. These vertices are 'candidates' because they are subject to an orientation test. The buffers also determine the neighbours data that is to be updated. Note that `nbuf[i]`, `cbuf[i]` actually determine an edge `e` of the potential neighbour of `*t` so the candidate edge of `*t` is `-e`.

The calling program is expected to set up and re-use the actual arguments for these buffers and hence they are declared the same size for both inserting members, although `InsertLSeg` only requires buffers of length 2. Examples of typical patterns of filling these buffers are given in the example code below and of §A.3.2. On returning, `Insert` *Y* sets buffers `vbuf[i]` and `nbuf[i]` to `NULL` as a convenience for the next use; `cbuf` is unchanged.

Determination of the candidate vertices for `*t`

For the `i`th edge of `*t`, the input allows up to 3 possible vertices for the candidate vertex opposite this edge. Using addition modulo 3 for `InsertTri` and modulo 2 for `InsertLSeg` these up to three possibilities are:

`vbuf[i]`

`vdestn[i]`=
the destination vertex of input edge `i+1`, (i.e. `nbuf[i+1]->VtxOppEdge[cbuf[i+1]+1]`.)

`vinit[i]`=
the origin vertex of input edge `i+2`, (i.e. `nbuf[i+2]->VtxOppEdge[cbuf[i+2]+2]`.)

If the input specifies exactly one non-`NULL` possibility for `t->VtxOppEdge[i]`, it is accepted as the candidate. If more than one non-`NULL` **MVert** pointer is present in the input and they are the same, then this redundantly specified vertex is used as the candidate. But if the multiple possibilities are not consistent, then `InsertTri` returns `FAILED`.
If no candidate is specified, then `InsertTri` returns `FAILED`.

Updating neighbours incidence data
If input `nbuf[i]` is a non-`NULL` pointer, then a validation check is undertaken to verify that `*t` and `nbuf[i]` can be identified as neighbours on edge `i`, and the complementary edge to edge `i`, respectively. If the check succeeds, the incidence data for both is updated. The principles of the validation check are described below and the details given in Appendix §A.1. This is the only way that `NULL` neighbouring data in a **MTri** or **MLSeg** can be modified.

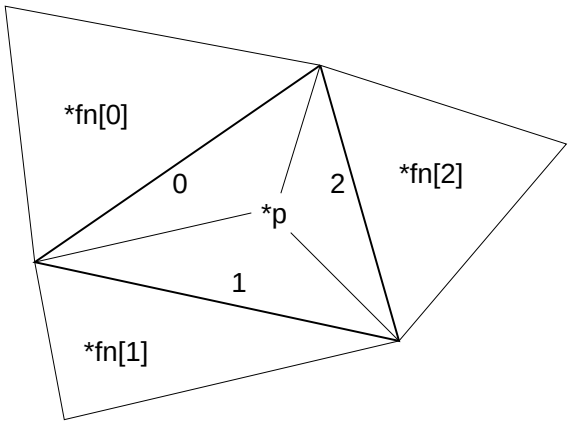
It will be apparent that it is important in coding with these classes that an object of type *Y* be removed with `Remove`*Y* **prior** to deleting it; otherwise unexpected `NULL` pointers are created in the other incident simple mesh objects and the list of the **Mesh2D** object. As we show in our simple example programs below, however, it is more common to remove an object and then reuse its space instead of deleting it. If a `Remove`*Y* operation results in an **MVert** with an empty adjacency list, then this isolated **MVert** is also deregistered from *this mesh* . This has the potential for a memory leak if the calling program does not retain a reference to the isolated vertex.

3.2 A simple example

As a simple, but reasonably typical, example of using these editing functions, we provide the following code for refining a triangle of a mesh into three by the simple insertion of a vertex. Assume that `MVert *p` has been defined as a point known to lie

inside triangle `*t`. We assume `*t` is registered in `Mesh2D *mesh`, but `*p` is not. The geometric configuration and labelling are shown in the figure adjacent to the code. `*t` is the central triangle and its local edge numbering is shown; the new triangles that have `*p` as a vertex are also shown.

```
// example : simple insertion of MVert *p in MTri *t

MTri **fn, *nbuf[3];   int cbuf[3], i;  MVert *vbuf[3];

fn = t->GetNOE();
    // fn[i] points to neighbour on edge i of *t

for( i = 0 ; i<3 ; i++)
 { vbuf[i] = NULL ; nbuf[i] = NULL; //  setting null pointers
   cbuf[i] = t->ComplEdgeNo(i);};

mesh->RemoveTri(t);
      // de-registers t from mesh

vbuf[0] = p; nbuf[0] = fn[0]; //  with cbuf[0], specifies t

mesh->InsertTri(t,vbuf,nbuf,cbuf);
     // *t is re-registered and *p is registered
     //  all incidences updated
     //  pointers in buffers  set to NULL; cbuf is unchanged

fn[0] = t ;  // saved for later

nbuf[0]= fn[1] ; cbuf[0] = cbuf[1] ;
nbuf[1] = t ;    cbuf[1] = 2 ;

t = new Tri(mesh-NewGTnum()) ;

mesh->InsertTri(t,vbuf,nbuf,cbuf);

nbuf[0]= fn[2] ;  cbuf[0] = cbuf[2] ;
nbuf[1] = t ;     cbuf[1] = 0 ;
nbuf[2] = fn[0] ; cbuf[2] = 1;

t = new Tri(mesh-NewGTnum()) ;
mesh->InsertTri(t,vbuf,nbuf,cbuf);
```



The two new **MTri** which must be added to the mesh are allocated memory using

a constructor `MTri( int gTnum)` whose parameter is the label of the new triangle. The argument used for this parameter in the example is the return value of the mesh **MTri** label server, `Mesh2D::NewGTnum()`. Note that for the final insertion, all three neighbours buffers are specified to ensure that their incidences are updated.

To simplify this example, we have not included tests of the `ErrCode` returned by the `Insert` function calls. This is a useful practice during code debugging.

The extension of this example to inserting triangles in a mesh to fill a hole that is star shaped with respect to ***p** is immediate. In particular, it is quite straight forward to extend this example to the Delaunay insertion technique described in George and Borouchaki [7] in which a set of triangles near the insertion point are removed producing a cavity that is star shaped.

The pattern of the operation of these editing functions is that they analyse their inputs for validity before undertaking their edits. If a validity check fails, or an edit fails at any stage, the mesh is returned to its initial state before the function returns, noting its failure. In the case of `InsertLSeg`, these edits may extend to data of objects registered in an adjacent mesh (`oppMesh`) of a composite mesh.

The concept is that `Insert`*Y* (and to a lesser extent, `Remove`*Y*) are guardians of a measure of mesh validity that is practical to check during editing. There are several motivations for centralizing this function in the **Mesh2D** class. Despite the relative complexity of the several editing tasks that occur, they are all logically related and we believe that programming simplicity is gained by limiting the ways in which logically related tasks can be accomplished. Another motivation is connected with the role of the mesh object classes in OO application programming. The **MVert** , **MTri** , **MLSeg** classes can be expanded by aplications programmers and become quite complex objects. Hence we have chosen to vest the understanding of mesh data in the **Mesh2D** class which is intended to be application independent.

These **Mesh2D** members functions incorporate some mesh validity rules i.e. are intelligent editors. As part of this overview, we present the principles of these rules as they apply to the input arguments, deferring the detailed specifications to Appendix §A.1.

candidate vertices

We have already described how the `Insert` routines determine the candidate vertices for the object to be inserted. These vertices must either be registered in *this* mesh, and in `oppMesh` if non-`NULL` , or eligible to be registered, i.e. not repeat a label of an already registered **MVert** , nor duplicate its coordinates.

neighbours incidences
The mesh in which the incidence data is to be edited should 'know' about the objects being edited. For `InsertTri` this simply means that non-`NULL nbuf[i]` should be registered in *this* mesh. The corresponding rule for `InsertLSeg` is more complex, although it can reasonably be inferred from the geometry. Recall that neighbours of a **MLSeg** must be of **MTri** type, and hence should only be registered in one **Mesh2D** of a composite mesh. Hence:

if `oppMesh` is `NULL`
  then at least one `nbuf[i]` must be `NULL` and a non-`NULL nbuf[i]` must be registered in *this* mesh.
else if `oppMesh` is *this* mesh
  then any non-`NULL nbuf[i]` should be registered in *this* mesh
else (a non-`NULL oppMesh` different from *this mesh* mesh, so)
  cannot have two non-`NULL nbuf[i]` both registered in *this* mesh or both registered in `oppMesh`

A second general principle is that if `nbuf[i]` is not `NULL` , then `nbuf[i]` $\rightarrow$ `GetNOE(cbuf[i])` must be `NULL` . This is intended to put safety ahead of convenience; i.e. it is based on the expectation that the presence of a non-`NULL` pointer in `nbuf[i]` $\rightarrow$ `GetNOE(cbuf[i])` is likely to signal a programming error.

Without question, these specifications are complex, and so is the task. Nevertheless, we feel that the operations are closely linked logically and that linking the validity checking and related editing in one task contributes both to programming simplicity, as argued above, and efficiency. In our programing with the **Mesh2D** class, we have found that the `Insert` functions are used with a few standard buffer patterns that become familiar. It is efficient to localize the mesh validity computations. Particularly in an OO programming scheme, much of the checking would have to be done in some form in any case.

As a general rule, we have tried to avoid including members of **Mesh2D** which are redundant, even if convenient. However, the edge swapping operation seems sufficiently basic to 2-D mesh programming that we have included it as an editing member function.

Debugging support
The **Mesh2D** class includes two elementary debugging facilities; a log file of mesh

operations and a formatted dump of the current mesh. A log file for a mesh object is created, or not, by the constructor with signature:

```
Mesh2D( int LogFileOn, const char* LogFileName )
```

Mesh2D has a member variable `LogFileOnIsOne` which can be used to control the recording of operations on *this* mesh subsequent to creating a log file. `LogFileOnIsOne` is initialized by the first argument of the constructor; if this argument is 1 (, or not zero) then a log file is created in the execution directory with name given by the second argument. If `LogFileOnIsOne` is subsequently set to 0, then the transcription of operations will be suspended until such time as it is reset to 1. If `LogFileOnIsOne` is initialized to 0 and subsequently reset to 1, an I/O error will occur, as might be expected.

The **Mesh2D** class dump member function has the signature:

```
void Dump(const char* FileName, const char* string)
```

A file with the name provided in the first argument is created (overwritten) and the second argument is entered in the first record as an identifying comment. The function then outputs the lists of labels of the **MTri, MVert, MLSeg** objects with their incidence data. Examples of log files and dumps are given in Appendix A.3.3.

4 The classes as software components for OOP

These classes are intended to be components in meshing applications software that uses an object oriented design. In Figure 10, we show a simple schematic of a conventional organization of software for a mesh generator plus an application, which is stiffness matrix generation. The schematic shows the user's view with circles representing (one of more) files and rectangles representing (collections of) programs. The mesh generation input files, labeled A, would contain a description of the geometry of the domain to be meshed, mesh control space data, etc. The stiffness matrix input file, labeled B, would contain physical model data, element shape data ... Typically, the computations of Figure 10, are performed on a project basis; e.g. a particular research project or the design/analysis cycle of a product. Many instances of the project defined model are to be computed. For purposes of discussion, we can identify

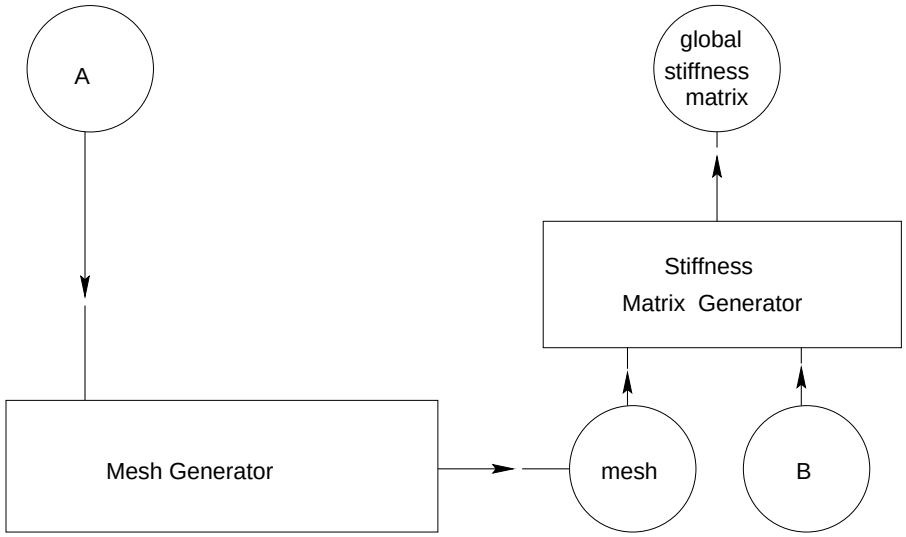


Figure 10: A conventional organization of mesh generation plus application software

the following three categories of information associated with the schematic:

the software design context This is the information about the class of inputs that the black boxes are intended to handle. It is found in the user's manuals, in the software specifications.

the project context This is the information that characterizes a specific project in the design context and is constant, or slowly varying, during the project.

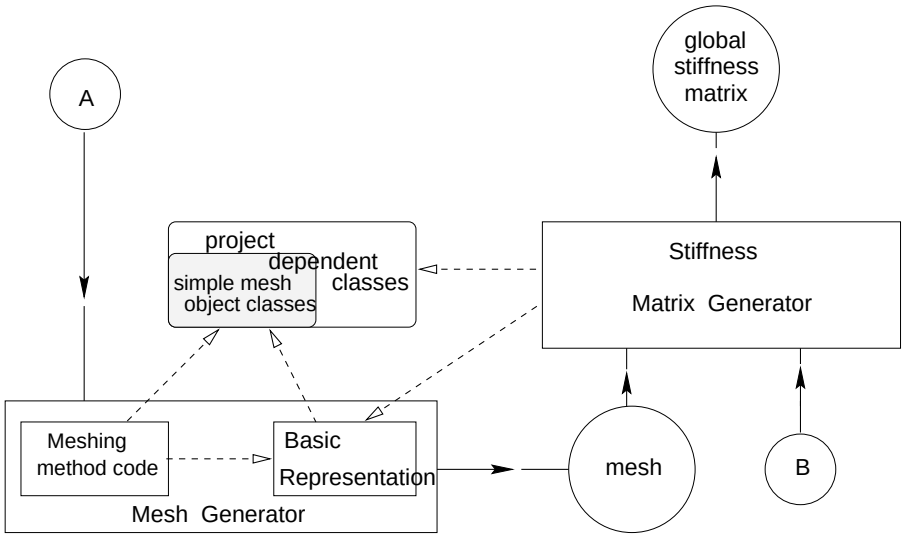


Figure 11: An OO organization of mesh generation plus application software

instance data the information specific to a particular run within the project.

The software design context describes the generality of the software. Clearly, the project context information must specify a subset of the software design context; in fact, this is its role. The project context information and the instance data combine to constitute the program data input of files A and B of Figure 10.

It is advantageous to design software with a significant breadth of generality, of course. However one result is that a larger component of project context information is needed in the input data for a specific project to identify the portion of of the software design context that it uses. Increasing generality presents the software designer with a several challenges; one being the anticipation of the details of all the projects to be covered and another being a suitable user interface. An object oriented software design can help to meet these challenges in a manner illustrated schematically in Figure 11. Here, a box representing classes that can be modified by the user to describe the project context information has been added to Figure 10; this box represents then project dependent classes that form part of the user interface. Since these classes reduce the component of project context information in the input data, the size of the circles for files A and B has been reduced. The mesh generation

software now includes a description of these project dependent classes as part of the the software design context. Naturally, this presupposes a relatively sophisticated 'user', which is the nature of software components. The challenge of anticipating the details of project usage becomes the design issue of finding suitable abstractions of the project context to require in a project dependent class definition.

The implication for the classes introduced above is that the members which the **Mesh2D** class expects of the simple mesh object classes must be present in the project dependent classes known to the mesh generator software. The project dependent classes must be named **MVert, MTri, MLSeg** and declare **Mesh2D** a friend. The dashed arrows of the figure point from software to classes on which it depends, i.e. that it knows about. The diagram does not distinguish, however, that the level of 'knowledge' of the simple mesh object classes differs between the mesh generation code and the **Mesh2D** class. The former know only the public members of the simple mesh object classes, in accordance with standard information hiding practices. The latter is implemented in terms of the private members since **Mesh2D** has been declared a friend by the simple mesh object classes.

The designer of the mesh generator now has the availability of project dependent classes to help expand the design context of this code. As a software component, the **Mesh2D** class does not constrain this context to any particular type of global coordinate system. It is unaware of whether **MVert** coordinates are pairs (2-D) or triples (3-D), whether the system is Cartesian or polar, Euclidean or Riemannian, nor what the arithmetic of the coordinate system is. A minimal extension of the basic mesh object classes that supports 3-D surface meshing is presented in the next section. Much of the mesh generation code can also be made coordinate system independent by the judicious use of geometric primitives that would be required of the project dependent classes. The triangulation of a simple polygonal face of a composite mesh, the conversion of a triangle to a Delaunay triangulation and Delaunay insertion of new vertices, decisions on local refinement can all be implemented using the **Mesh2D** class plus simple abstractions of geometric information that are coordinate system independent. The ear clipping method , O'Rourke [12] requires a line segment intersection primitive; Delaunay conversion and insertion require an InCircle test, Guibas and Stolfi, [8]; local refinement requires some quantitative data such as the length of the longest edge, or the radius of the circumcircle, Chew [3]. In Figures 12 and 13, we show two outputs from testing Delaunay insertion in a composite mesh. Edges that are line segments are shown with darker lines. The data were generated with two different main programs and **M***X* classes , one with

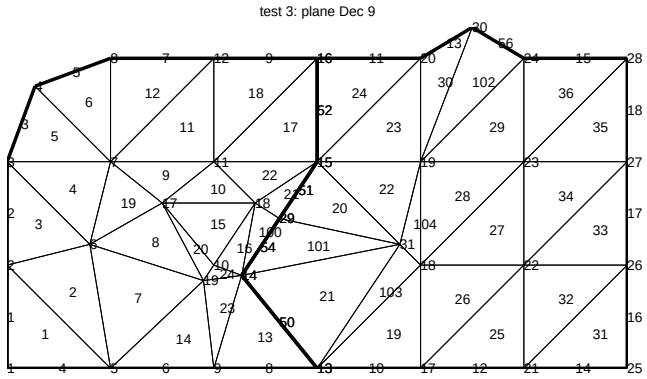


Figure 12: Delaunay insertion using plane project dependent classes

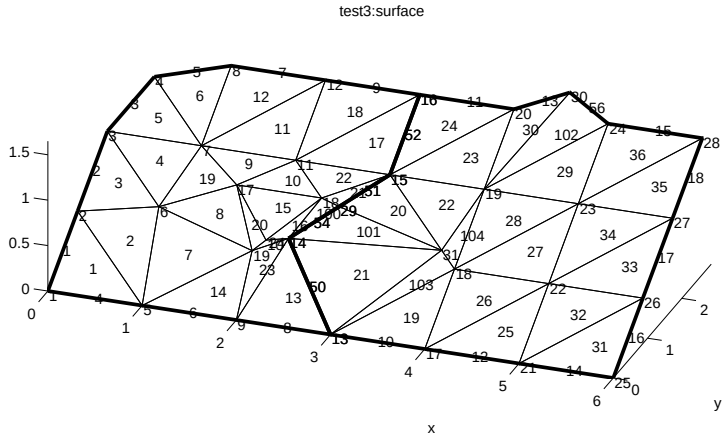


Figure 13: Delaunay insertion using surface project dependent classes

a 2-D and one with a 3-D global coordinate system, but each linked with the same Delaunay insertion code and **Mesh2D** implementation.

There is a literature on OOP for the finite element method which is based on this design strategy applied to stiffness matrix generation software, Mackie, 1992 [10], revue européene des Eléments finis, 1998 [2] . The result is to generalize the stiffness matrix generation code to apply to a broad range of physical models and

element types, which the model and element type for a specific project determined by project dependent class. In Figure 11, we have shown the mesh generator and the stiffness matrix generator sharing the same project context classes as well as both using the **Mesh2D** class. In this context, the project dependent classes would be expected to carry data for the stiffness matrix generator as well; physical model data, boundary conditions, element shape function data. They would become substantial objects; double inheritance involving basic objects classes for the stiffness matrix code is a plausible mechanism for this extension. The 'users' for this interface then would be expected to be developing FEM software or at least sophisticated programmers. This distinction between the different roles between the simple mesh object classes and the **Mesh2D** class has contributed to the motivation for making the former simple, relatively passive classes and concentrating the mesh knowledge in the **Mesh2D** class, with its intelligent editing functions. It also provides the motivation for isolating knowledge of the mesh coordinate system in the simple object classes.

How can we ensure that the project dependent classes include the members required by the **Mesh2D** class embedded in the mesh generator, or mesh application software, i.e. the stiffness matrix generator as shown in Figure 11? The primary OO mechanism for this is to define base classes which we will designate **BVert** , **BTri** , **BLSeg** , that have the members that we have attributed to the simple mesh object classes in §2, and allow the project context classes, now designated **MVert, MTri, MLSeg** , to be derived from them. This elegant approach is unfortunately prone to run time inefficiencies, [5]. It is, however, simple to implement and has been the mode of our initial prototyping experiments with these classes. It is used for code sharing only in this context, no use of polymorphism is implied by this design.

On a project basis, it would be feasible to simply expand the source code of the **B***X* classes to add the additional project context information. We have done this for some performance studies ; however, it would lead to software management problems when maintaining the mesh generator or mesh application software.

Effective, if not simple, techniques for this purpose can be based on template programming. The CGAL library ,[5] and §6, makes extensive use of this approach. A general mechanism for converting object hierarchies to template classes has been described by Weihe, 1998, [15]. The basic idea is to write template classes that use the members of the **B***X* classes as template features as well as carrying other project data. These template classes would then be instantiated with the simple object classes to provide the actual project context classes.

Separate Compilation If the **M***X* classes are to be project dependent and the **Mesh2D** class not, it is clearly desirable to be able to compile the **Mesh2D** class indepen-

dently of the **B***X* classes. `C++` inheritance provides a mechanism for this. To use it directly, the **Mesh2D** class would be declared in terms of the **B***X* classes which declares some of its function members virtual and the **M***X* classes would be necessarily derived from the **B***X* classes. However, we do not want to be tied to inheritance, so we have employed a slightly more complicated approach that permits independently declared **M***X* classes. For this reason, the **Mesh2D** class has been written directly in terms of the **M***X* classes and only uses pointers to objects of these classes. In particular, it neither creates nor deletes memory space for the objects themselves.

This approach requires some care in the organization of header files and the storage of code, which we will illustrate with the help of the appendices. A header file that declares a set of (possibly project dependent) **M***X* classes will be named `MVandT.h` (see Appendix §A.3 §A.3.1 for an example.). Each header files starts with a conditional compilation check on an identifier for the file, e.g. `#ifndef _MVandT_H_` [3] for `MVandT.h`. The implementation of the **Mesh2D** class can be compiled with a simple minimal declaration of the **M***X* classes into an object library or semi-linked object module. In the the code of the appendices, this skeleton set of **M***X* classes are derived from the base class declared in `BVandT.h` and shown in Appendix §A.2.1. that declares the minimal members required by the **Mesh2D** class. The **Mesh2D** class header file gets this skeleton declaration by an inclusion, typically `#include "MVandT.h"` from the current directory. Note that the **B***X* class member functions are written in terms of the **M***X* class objects, and `BVandT.h` starts with a forward reference to these classes.

Independently, each set of project dependent simple mesh object class implementations, along with other project dependent source, and a `main` program, can be compiled using the `MVandT.h` declarations appropriate to the project. The resulting object code can then be linked with the object code for the **Mesh2D** class implementation.

Details of the directory structure for this technique can be illustrated using the tutorial example in Appendix §A.3. Consider a Unix directory, `source`, that contains:

- `BVandT.h`,
- a skeleton `MVandT.h` derived from it
- `Mesh2D.h` and the source for its implementation.

Suppose further that we have several subdirectories; one for each project, and one of these, named `source/surface` contains the example of Appendix §A.3. The `source/surface/MVandT.h` file is shown in §A.3.1. In the subsequent section of this Appendix, the code for the main program is given, also stored in `source/surface`.

[3] see Ellis and Stroustrup, Chapter 16, [4]

The opening lines illustrate a key step; the `MVandT.h` header is included *before* the `../Mesh2D.h` header file. As a result, the `_MVandT_H_` identifier has already been set as defined when the inclusion of `../Mesh2D.h` occurs. Consequently, the declarations in the `../MVandT.h` header file included by `../Mesh2D.h` are conditionally skipped in the compilation and it proceeds using the project dependent class declarations. The advantage of this complexity is that we are free to use inheritance via the $\mathbf{B}X$ classes or nor as we choose.

5 The project dependent classes: a tutorial example

In this section, we discuss a minimal geometric extension of the basic simple mesh object classes of §3 that can support 3-D composite mesh representation. We do this by means of a example main program that is small enough that the code can be included in Appendix §A.3. Although it is trivial in its extent, the program does illustrate a general technique for generating an initial triangulation on a polyhedral surface. Figure 14 shows two views of a boundary representation of a simple trough. The vertices of the trough are labeled 1 to 6 (in Helvetica font) and the line segments

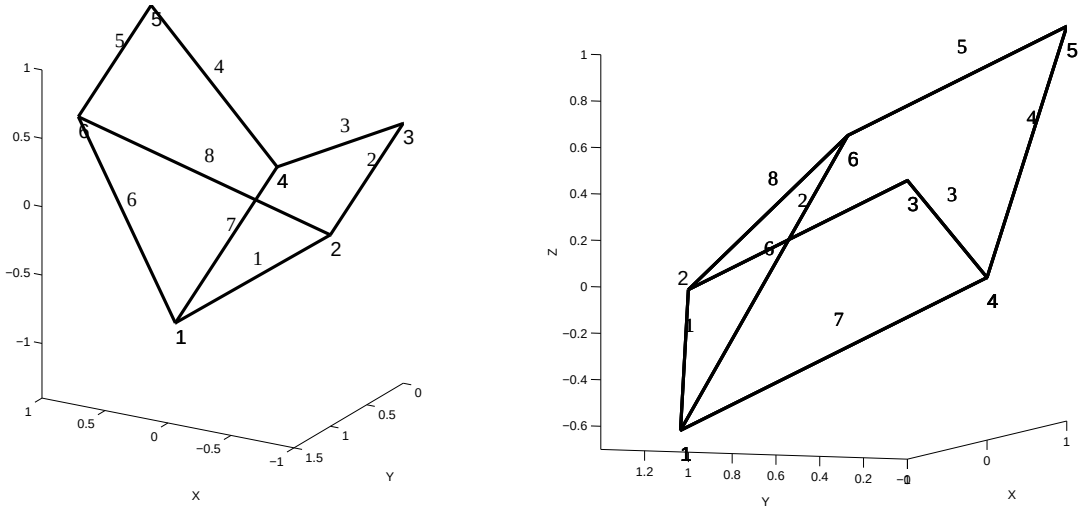


Figure 14: Boundry representation of simple trough example: 2 views

are labelled 1 to 8 (in Times Roman.) The three plane sides of the trough are the quadrilaterals, v_1, v_2, v_3, v_4, v_1, v_4, v_5, v_6, and the triangle v_1, v_2, v_6. The trough is open at the top; vertices v_2 and v_4 are the lowest vertices on this open boundary, at height $z = 0.0$. If water were poured into the trough, it would fill until the water spilled out at v_2 and v_4.

The **MVert** class used for this example extends **BVert** by adding the following members:

- (x, y, z) Euclidean coordinates of type `double`
- an implementation of the virtual `CoordDiff` member, which defines the conditions under which the coordinates of two **MVert** are regarded as different

- a new function , `Write()` to output the data for a vertex to `stdout`.

(See Appendix §A.3.1 for details.)

The main geometric extensions of **BTri** that are present in the **MTri** class of this example are connected with confirming the correct orientation of candidate vertices for `Mesh2D::InsertTri`. The class declares a static member `static double MeshPlanePos[3]` which provides a single array space that can be used as a buffer by any **MTri** object. The intention is that when `MTri *t` is to be inserted into `Mesh2D mesh`, a positive direction for `mesh` should be recorded in `t->MeshPlanPos`; let it be denoted $\vec{n}_{pos}$. The implementation of `MTri::Orientation` then checks that the positive normal to the triangle, T, defined by the candidate vertices $v_i^{(cand)}$ of `*t` points to positive side of `mesh` by computing the inner product

$$< \vec{n}_{pos}, (v_1^{(cand)} - v_0^{(cand)}) \times (v_2^{(cand)} - v_0^{(cand)}) > \qquad (2)$$

Remarks 1. Note that **Mesh2D** simply enables this validity check to be used. A program that uses **Mesh2D** can by-pass it by providing an **MTri** class with an `Orientation` member function that always returns 1, for example.
2. As we illustrate in this example, the flexibility in this technique to select any positive direction for a mesh plane simplifies the preparation of the input for the object to be meshed. The inner product (2) is related to the signed area of *this* **MTri** as seen from the mesh positive direction. For complex objects involving triangles that are less well shaped than our simple example, the difficulty of correctly computing the sign of this projected area may be an issue. For robust implementations, it would be preferable to the the actual mesh plane unit normal as the `MeshPlanePos` vector.
3. We note that for 2-D meshes, the `MTri::Orientation` member function can be implemented simply as a signed triangle area computation.

Appendix §, A.3.2 provides a program that constructs a composite mesh for this trough represented by array `Mesh2D *mesh[3]`. The basic pattern of the program is that of a general approach to constructing an initial triangulation of a polyhedral surface:

step 1 define a vertex coordinate array (more generally by reading from a file)

step 2 build a boundary representation of the composite mesh

step 3 triangulate each mesh of the composite mesh

In step 2, both the vertices and the line segments are registered in their respective meshes. Note that the line segments for the internal edges are registered with `NULL` neighbours in both meshes. This facilitates the triangulation step.

In step 3, the first triangle to be inserted in `mesh[i]` must fill the **MTri** static buffer `MeshPlanPos` with a positive direction for `mesh[i]`. In this example, the positive sides of the planes are identified with the inside of the trough. The program shows very simple choices of positive directions for each; i.e. the positive x axis for `mesh[0]`, the negative x axis for `mesh[1]`, the negative y axis for `mesh[2]`.

The log file for each of the `mesh[i]` objects has been created; When the trough has been constructed, a description of the incidence data for each mesh is printed out using the `Mesh2d:Dump` member. Note that the recording of transactions in these files has been turned off just prior to thes dumps. The log files for $i = 0$ and 2 and the dump file for $i = 0$ are shown in Appendix §A.3.3.

Remarks 1. For the first insertion, the return value of `InsertLSeg` has been checked, but not subsequently. We note that it is a useful practice in developing programs using **Mesh2D** to include such tests during initial debugging runs. Then, once confidence has been gained in the code, it may be worth removing these tests in the interests of readability and, in the case of heavily used code sections, efficiency.
2. The trough could be represented by a single mesh object; i.e. it is not essential to use a three mesh composite mesh. The view of the trough from the centroid of v_3, v_4, v_5 shows the positive side of all three faces, Hence a single `MeshPlanePos` direction could be assigned to a **Mesh2D** object, `Mesh2D *meshTrough`, that could serve the `MTri:Orientation`) member of all five **MTri** ; e.g. the direction from v_1 to the centroid. The internal line segments, (numbered 1,6, and 7), could be omitted and neighbouring incidences to neighbouring **MTri** on adjacent faces created. The resulting single mesh would be topologically equivalent to the planar mesh created by projecting the trough onto the $y = 0$ plane along the mesh plane positive direction.

6 Related Work and Conclusion

A number of the issues and approaches discussed in this report have been considered, with varying degrees of explicitness, by other authors. Comments on the literature in the OOP approach to the FEM and its relation to this work have already been been included in §4.

The CGAL project [4] is producing a comprehensive library of computational geometry algorithms for applications. The report by Fabri, et al , 1998, [5], gives an overview of the programming strategies employed as well as the project goals and management. The documentation web site for the project

`http://www.cs.uu.nl/CGAL/Information/doc_html/index.html`

provides explicit descriptions of library and class details. The simple mesh object classes of this report are related to the elementary objects of the triangulation and half edge classes of CGAL. The **Mesh2D** class is a specific container class while the CGAL strategy is to support implementations that are generic with respect to the container classes. CGAL has a clearly defined strategy for these generic implementations that is based on template programming. It would appear that this strategy is meant to be used on a project basis, as well as for the development of geometry specific codes for the library itself.

In [11], Mobley, Carroll, and Canann comment on the benefits of separating geometric from incidence data in their design of finite element oriented classes for software that interacts with both CAD data and finite element mesh data for 2 and 3 dimensions. Several authors have described designs and implementations which broaden the design context of the mesh list class further by parametrizing the topology of the mesh elements. The CHAINS code of Palmer '95 [13] and the QMG code of Vavasis [5] use k-simplicies as the basic element; Vavasis also uses a general object class called a face which is parametrized by its intrinsic dimension and its embedded dimension and has some similarities to the **MLSeg** class of this paper when these dimensions are 1 and 2 respectively.

The report has tried to provide the option to be read with more or less detail by giving a combination of general overview and specific detail of the classes presented. We have described their intended role in an established OOP approach to mesh applications software and indicated by tutorial examples the style of programming that they support. Hopefully, this will indicated how we intend the classes to provide for

[4] `http://www.cs.uu.nl/CGAL/`

[5] see `http://simon.cs.cornell.edu/home/vavasis/qmg-home.html`

simplifying mesh programming while facilitating the generality of method implementations. A significant part of the simplification is derived from centralizing the mesh data editing in the four editing function members of the **Mesh2D** class. We have not seen a similar strategy in other class desriptions. The separation of incidence data from coordinate based geometric data is a common approach to facilitating the generality of method implementations and is present in one form or another in the designs mentioned above. The classes of this report make this separation in the project dependent mesh object classes; the separation may be explicit (by a inheritance hierarchy, or template instantiation) or implicit, as suits a particular project.

A Appendices

A.1 Specifications for Mesh2D editing members

The signature for `InsertTri` is

```
ErrCode InsertTri(MTri *t, MVert* vbuf[3], MTri* nbuf[3], int
                                cbuf[3]);
```

The signature for `InsertLSeg` is the same except for an additional parameter `Mesh2D* oppMesh`
Both of these member functions return the logical flag `ErrCode` defines in `Global.h` with values `SUCCEEDED` for a sucessful insertion, `FAILED` otherwise.

The basic principles of the design of this signature are described in §3; here we give a detailed list of specifications for these functions.
Requirements Common to `InsertTri , InsertLSeg`

Insert.1 `*t` is the object to be inserted, or registered, in *this mesh* . `t->GTnum` must be valid[6] for *this mesh* , but `t` must *not* be registered in *this mesh* . Any existing `VtxOppEdge, NaybrsOnEdge` data for `*t` will be ignored.

Insert.2 `vbuf[i]` is either `NULL` , or points to a potential candidate for `VtxOppEdge[i]` of `*t`. `vbuf[i]` need not be registered in *this mesh* , but `vbuf[i]->GVnum` must be valid for *this mesh* . Morever, the coordinates of this vertex must be distinct from any other vertex registered in the mesh, (based on the **MVert** member function `CoordDiff`).

Insert.3 Either `nbuf[i]` is `NULL` , or the pair `nbuf[i],cbuf[i]` should determine an edge in *this mesh* . In this latter case, edge `cbuf[i]` of `nbuf[i]`) is a candidate for the `i`th edge of `*t`(reversing the direction) *AND* `nbuf->NaybrOnEdge[cbuf[i]]` must be `NULL` . The value of `cbuf[i]` must be consistent with the actual type of `*t` i.e. 0,1 for **MLSeg** and 0,1,or 2 for **MTri** .

Insert.4 There is a complex requirement governing the acceptable combinations of `NULL` input for `vbuf`, `nbuf` described in the following paragraph for determining vertices for `*t`

Strictly speaking, these requirements apply for `i = 0,1,2` for `InsertTri` and `i = 0,1` only for `InsertLSeg`.

[6] It must be positive and not duplicate an existing **MTri** registered in this mesh

Determination of the candidate vertices for `*t`
For the `i`th edge of `*t`, the input allows up to 3 possible candidate vertices for the vertex opposite this edge (i.e. `t->VtxOppEdge[i]`).
I.e. (using addition modulo 3 for `InsertTri` and modulo 2 for `InsertLSeg` in subsequent expressions)

`vbuf[i]`

`vdestn[i]`=
the final vertex of candidate edge `i+1`, (i.e. `nbuf[i+1]->VtxOppEdge[cbuf[i+1]+1]`.)

`vinit[i]`=
the starting vertex of candidate edge `i+2`, (i.e. `nbuf[i+2]->VtxOppEdge[cbuf[i+2]+2]`.)

If the input specifies exactly one candidate for `t->VtxOppEdge[i]`, it is accepted as the candidate. If more than one vertex specification is present in the input and they are consistent, i.e. the global **MVert** number of each is the same, then this redundantly specified vertex is used as the candidate. But if the multiple specifications are not consistent, then `InsertTri` returns `FAILED`.
If no candidate is specified, then `InsertTri` returns `FAILED`.
Note: Non-`NULL` `nbuf` data is used both for identifying mesh neighbours to be updated for `*t`, and for specifying candidate vertices for it. In fact, for some insertions, it is the need for the former purpose that leads to redundancy in the second as discussed.
Note: candidate vertices are determined for *this mesh*

Requirements Specific to `InsertTri`
For a successful insert:

InsertTri.5 Non-`NULL` `*nbuf[i]` may be either **MTri** or **MLSeg** . and be registered in *this mesh*

InsertTri.6 The **MTri** member function `Area` must evaluate positive for the the candidate **MVert** s

Requirements Specific to `InsertLSeg`
For a successful insert:

InsertLseg.5 Requirements **Insert.1** must also apply to `oppMesh`.

InsertLSeg.6 for non `NULL oppMesh`, candidate vertices must be registerable in `oppMesh`

InsertLSeg.7 non-`NULL nbuf[i]` must be **MTri** , i.e. cannot be **MLSeg**

InsertLSeg.8 if exactly one of `nbuf[i]` is `NULL` , i = 0,1, then the non-`NULL nbuf` must be registered in *this mesh*. `oppMesh` may be `NULL` or not.

InsertLSeg.9 if both `nbuf[i]` are not `NULL` , then `oppMesh` must be non-`NULL`. One of the `nbuf[i]` must be registered in *this mesh* and the other in `oppMesh` (including the case `oppMesh` = *this mesh*.)

Note: `vbuf[2]`, `nbuf[2]` and `cbuf[2]` are ignored.

Updating common to `InsertTri`, `InsertLSeg`
For a successful insertion:

Update.1 Candidate vertices are assigned to `t->VtxOppEdge` and registered in *this mesh* (in *Vertices*).This is the only way to assign vertices to **MTri, MLSeg** .

Update.2 `t` is registered in *this mesh* (in *Triangles*).

Update.3 `t->NaybrOnEdge[i] = nbuf[i]` (whether `NULL` or not.)

Update.4 if `nbuf[i]` is not `NULL` then `nbuf[i]->NaybrOnEdge[cbuf[i]]` is set to `t`. These are the only ways to assign neighbours data for an **MTri, MLSeg** .

Note: For `InsertLSeg`, **Update.3** and **Update.4** apply regardless of which meshes are involved.

Updating specific to `InsertLSeg`

UpdateLSeg.5 **Update.1** and **Update.2** also apply to non-`NULL oppMesh`

UpdateLSeg.6 If `nbuf[i]` is in *this mesh* then `t->MeshOnEdge[i] = this`. But if `nbuf[i]` is in `oppMesh` then `t->MeshOnEdge[i] = oppMesh`.

In particular, **MVert** can be registered in *this mesh* only through `Insert`X member functions, although not necessarily only through those of *this mesh*, i.e. possibly through a neighbouring mesh.

On return
The buffers `vbuf` and `nbuf` are set to `NULL` to facilitate subsequent updates. On returning `ErrCode == FAILED`, no mesh data is changed.

`RemoveTri , RemoveLSeg`
The signatures of these **Mesh2D** member functions are :

```
ErrCode RemoveTri( MTri *t ); ErrCode RemoveLSeg (MLSeg *t);
```

Updates common to both

Remove.1 removes ***t** from *Triangles* (or *BoundaryRefs*) for this mesh.

Remove.2 all neighbours pointers in ***t** are set to `NULL` and all pointers to ***t** in its neighbouring mesh entities are set to `NULL` .

Remove.3 removes **t** from the incidence list of each of its vertices. If, for one of these vertices, this results in an incidence list with no **MTri** or **MLSeg** objects registered in this mesh, then the vertex is deregistered in this mesh.

Note that **Remove***X* does not delete *X* which is the task of the destructor for *X*. This remark also applies to any **MVert** objects that are deregistered implicitly. This has some potential to be a source of a programming bug in which a vertex becomes de-registered from every mesh and is assumed 'gone' by the programmer, but has not actually been deleted from dynamic memory.

A.2 Header files

This appendix contains:

a) the header file `BVandT.h` for the **BVert, BTri** , and **BLSeg** that define the minimal simple mesh object class members expected by **Mesh2D**

b) a header file `MVandT.h` that derives **MVert MTri** , and **MLSeg** classes from **B***X* and that can be used to compile the **Mesh2D** class implementation

c) the header file declaring the **Mesh2D** class

An example of project dependent classes derived from the **BVert, BTri** , and **BLSeg** classes and implementations of them are given in the example code of the next appendix.

A.2.1 the BVert, BTri , and BLSeg class header files

```
#ifndef _BVandT_H_
#define _BVandT_H_
#include <iostream.h>     // defines NULL   (!!)

// Forward references needed by BTri, BVert and BLSeg
class Mesh2D;  class MTri; class MVert; class MLSeg;

// Structure used by the adjacency list for each BVert
struct BTLSegList {      // Triangle linked list for vertex adjacency info
  MTri *t;        // A triangle or line segment pointer
  BTLSegList *next;       // Next structure in the list
};

////////////////////////////////////////////
//               Class BVert

class BVert {
  friend  Mesh2D ;

  public:
   int GVnum;        //  the global vertex  label
   virtual int CoordDiff(MVert *w) = 0;
        // compares coords of this BVert with *w
        // defined as pure virtual to force  subclass definition
        // returns 0 if coordinates are, by definition of this function, the same
```

```
//         1 ( not zero) if they are different

  protected:
    BVert();          // default constructor
    BVert(int num);   // explicit constructor providing global label
    virtual ~BVert();                // default null destructor
    BTLSegList *adjList; // triangles or line segments incident on this vertex

}; // BVert

////////////////////////////////////////////////////////////////////////////
// Class BTri
//

class BTri {
  // triangle data can be accessed by the members of this class
  // but not modified.   Modification of triangle  incidence data can
  // only be done through the Mesh2D class member functions
  friend Mesh2D;

  public:
    int GTnum;  // global Tri index
    MVert *GetVOE(int edgeNo);
      //  if -1< edgeNo <3
      //   then  returns pointer to vertex opposite edge of edge
      //             numbered edgeNo
      //   else returns NULL
    MVert **GetVOE();
      //  returns pointer to a variable array of MVert opposite edges
    MTri *GetNOE(int edgeNo);
      // requires -1< edgeNo <3 or program terminates !!
      // if -1< edgeNo <3
      //   then  returns pointer to neighbour (MTri or MLSeg) on edge
      //             numbered edgeNo
      //   else returns NULL
    MTri **GetNOE();
      //  returns pointer to  variable array of neighbours on edges
    MTri *GetNOE( int edgeNo, int &cegno);
     // returns neighbour on edge edgeNo AND complementary edge no
    virtual double Orientation(  MVert** vbuf )  = 0;
     // returns pos if vbuf[i], i=0,1,2 form a counter clockwise triangle
     // viewed from positive side of a given mesh plane
```

```
    //  For 2-D, treat coordinate plane as embedded in 3-D with  pos
    //  vertical as positive direction
    //  (Note: virtual function initializer must be 0!)
   int  ComplEdgeNum(int edgeNum ) ;
     //  if (-1 < edgeNum < 3)
     //   then
     //     if  edge(*t,edgeNum) has NULL neighbour opposite this tri
     //      then the function  returns  -1
     //      else Let neigh be neighbour of this tri on edge(*t,edgeNum)
     //           The function returns edgeNum2 such that
     //                    e = edge(*t,edgeNum) = -edge(*neigh,edgeNum2)
     //   else  the function returns 3

  protected:
    BTri();                              // default constructor
    BTri(int gTnum);                     // explicit constructor
    ~BTri() {}     // destructor and constructor are protected
    MTri *NaybrOnEdge[3];
      // pointers to MTri and MLSeg mesh neighbours for each triangle edge
    MVert *VtxOppEdge[3];
      // pointers to  MVert opposite  each triangle edge
}; // BTri


////////////////////////////////////////////////////////////////////////////
// Class BLSeg
//

class BLSeg: public BTri{   // for a BLSeg, VtxOppEdge[2] == NULL
  friend Mesh2D;

  public:
    Mesh2D *GetMOE(int edgeNo);
      //if  -1 <edgeNo <2
      // then returns pointer to mesh (possibly NULL)  sharing edge edgeNo
      // else  returns NULL
  protected:
    BLSeg(): BTri() {} ;
    BLSeg(int gTnum): BTri(gTnum){};
    ~BLSeg() {}
    Mesh2D *MeshOnEdge[2];
        // pointers to Mesh2D mesh  on edge 1
```

```
}; // BLSeg

#endif  // _BVandT_H
```

A.2.2 the MVert, MTri, and MLSeg class header files

```
#ifndef _MVandT_H_
#define _MVandT_H_

#include "BVandT.h"

class MVert : public BVert {
  public:
   double X,Y;             // Euclidean plane coordinates
   MVert();                      // default constructor
   MVert(int GVnum, double x, double y);             // explicit constructor
   ~MVert(){} ;                  // default null destructor
   int CoordDiff(MVert *w) ;     // required by Mesh2D
};

class MTri : public BTri {
  friend class Mesh2D ;

  public :
   MTri(): BTri() {}  ;                       // default constructor
   MTri(int gTnum): BTri(gTnum) {}; // explicit constructor
   ~MTri() ;                      // default destructor
   double Orientation( MVert**  vbuf); // signed area of triangle with
} ;

class MLSeg : public BLSeg {
  friend class Mesh2D ;

  public:
   MLSeg(): BLSeg() {};
   MLSeg(int gTnum): BLSeg(gTnum) {};
   ~MLSeg(){} ;
   double Orientation(MVert** vbuf);
};

#endif //   _MVandT_H
```

A.2.3 The Mesh2D header file

```
#ifndef _Mesh2D_h_
#define _Mesh2D_h_

#include <fstream.h>
#include "Hash.h"       // includes declaration of basic Hash table
#include "MVandT.h"

enum ErrCode { SUCCEEDED = 0, FALSE = 0, FAILED = 1, TRUE = 1 };

class Mesh2D {

//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// This class is a container class for storing lists of vertices,
// triangles, and boundary line segments.  These lists will be referred
// to as Vertices, Triangles and Boundaries, respectively.
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

public:
  Mesh2D( int LogFileOn, const char* LogFileName) ;
    // constructor
    // if LogFileOn = 1 (, or not zero,)
   //   then create and open for write a log file named LogFileName
   //        in the directory of executing program
  int LogFileOnIsOne ; // flag to control writing entries in LogFile
                       //  On if =1 (or not 0) ; Off if = 0
  ~Mesh2D();

  ////////////////////////////////////////////////////////////////////
  //   **  ACCESS ROUTINES **
  ////////////////////////////////////////////////////////////////////

  int NewGVnum(){return ++NewVertex;};
  int NewGTnum(){return ++NewTriangle;};
  int NewGLSnum(){return ++NewLSeg;} ;
    // returns a global vertex,triangle, or line segment number
    // larger than any EVER registered in this mesh
    //  NOTE: NewX may also be modified by InsertY  Y = Tri, LSeg

  int NumVtx() { return NVtx;};
  int NumTri() { return NTri;};
  int NumLSeg() { return NLSeg;};
```

```
    // return the number of vertices, triangles, boundaries

  MVert *FirstVtx();
  MTri *FirstTri();
  MLSeg *FirstLSeg();
    // initializes a scan of Vertices, or Triangles, or Boundary Refs
    // ensures   return of a pointer to an MVert, or MTri, or MLSeg
    // ; or NULL if  the corresponding list is empty

  MVert *NextVtx();
  MTri *NextTri();
  MLSeg *NextLSeg();
    // returns the next MVert in a scan of Vertices
    //         ( or next MTri , or next MLSeg)
    // requires  a prior call to FirstX or NextX that
    //           returned non NULL pointer, for X = Vtx , or Tri, or LSeg
    // ensures   return  of either a pointer to an MVert(,MTri, MLSeg) or NULL
    //           if current scan is complete

  MVert *VtxOfI( int vNum );
  MTri *TriOfI( int tNum );
  MLSeg *LSegOfI( int tNum );
    // ensures   return of pointer to MVert with index vNum
    //           (or MTri with index tNum , MLSeg with index tNum)
    //           NULL if not in Vertices(Triangles, Boundary Refs)

  int Mesh2D::TriOnVtx(MVert *v, MTri ** tList, int maxNumOfTri) ;
    //  requires: vertex *v and array MTri * tList[maxNumOfTri] declared in
    //            calling program. maxNumOfTri must be positive
    //  ensures: function value is total number of MTri objects (including MLSeg)
    //          incident on v in all meshes. The first min(TriOnVtx, maxNumOfTri) of
    //          them are stored in array tList[k].
    //      If any of the MTri are registered in this mesh, then
    //          tList[0] is in this mesh

  inline ErrCode Mesh2D::IsBoundary( MTri *t)
{ //  returns TRUE if *t is MLSeg - FALSE otherwise
        if(t==NULL) return FALSE;
         else
          return (t->VtxOppEdge[2] == NULL ? TRUE: FALSE);
};
```

```
inline ErrCode InThisMesh(MTri *t) ;
 // returns TRUE if *t is registered in this mesh

void Dump(const char* FileName, const char* title);
  // Debugging aid
  // Dump the contents of the mesh to FileName
  //  with title included at top of dump file

//////////////////////////////////////////////////////////////////
//  **  UPDATE ROUTINES **
//////////////////////////////////////////////////////////////////

// the data insertion members,  InsertTri and InsertLSeg, perform several
// types of data updating after some validation checks
// Assuming the validations are passed, these members :
//  i) register *t and possibly new MVerts in this mesh
//  ii) update incidence data in *t about its vertices and  neighbours
//  iii)  update the reciprocal incidence data in the vertices and neighbours
//
//  Similarly, the data removal members, RemoveTri and RemoveLSeg, perform
//  several modifications of the incidence data for the mesh and its objects
//  after some validations. I.e.
//  i)   removal of t from the  Triangles or Boundaries  lists and the
//          adjacency list of each of its vertices
//  ii)  NOT the deletion of the  dynamic  memory allocated to *t
//  iii) the removal of t as a neighbour of its  pre-removal neighbours
//  iv)  the removal of any vertex of *t which has an empty adjacency list
//       after removing t from Vertices list; but not the deletion of
//       the dynamic memory allocated to this vertex.
//
//  ****   See Tech Report Appendix A for details ****

ErrCode InsertTri(MTri *t, MVert* vbuf[3], MTri* nbuf[3], int cbuf[3]);

ErrCode InsertLSeg(MLSeg *t,  MVert* vbuf[3], MTri* nbuf[3],
                                    int cbuf[3], Mesh2D* oppMesh);

ErrCode RemoveTri( MTri *t );

ErrCode RemoveLSeg (MLSeg *t);

ErrCode Mesh2D::EdgeSwap( MTri *t, int locENum);
```

```
        //  Let *t1 = neighbour of *t on  edge determined by *t and locENum
  //  requires   *t and *t1 both be MTri  ; No test to be sure each is in
  //             this mesh
  //   returns SUCCEEDED if  it swapped  edge determined by *t and locENum
  //         for alternative diagonal  in quadrilateral formed by  *t, *t1
  //         no convexity check on this quadrilateral
  //   returns FAILED otherwise

protected:

  ofstream LogFile;    // name of the log file
  ofstream DumpFile;   // name of the debugging dump file
  int NVtx;            // number of vertices in Vertices
  int NTri;            // number of triangles in Triangles
  int NLSeg;           // number of line segments Boundaries
  int NewVertex;       // number of the next vertex (always increasing!)
  int NewTriangle;     // number of the next triangle (always increasing!)
  int NewLSeg;         // number of the next LSeg (always increasing!)

  HashTable<MVert> *vHTable;     // hash table for Vertices
  HashTable<MTri>  *tHTable;     // hash table for Triangles
  HashTable<MLSeg> *bHTable;     // hash table for Boundary Line Segs

private:

  ErrCode RemoveVtx( MVert *v );
  ErrCode CheckVertices(MTri *t, int dimT, MVert* vbuf[3], MTri **nbuf,
        int *cbuf, int* alreadyIn);
  ErrCode CheckAndUpdateNaybrsLSeg(MLSeg* t, MVert **vtxOppEdge,
                                   MTri **naybrOnEdge);
  ErrCode AddVtx( MVert *v);
  ErrCode AddAdjacency(MVert *v, MTri *t);
  ErrCode DelAdjacency(MVert *v, MTri *t);
};  // Mesh2D

#endif   // _Mesh2D_h_
```

A.3 Tutorial example - a trivial trough

Figure 14 shows two views of the boundary representation of a simple trough with three planar faces. The code that constructs a representation using an array `Mesh2D* mesh[3]`, and the **MVert, MTri, MLSeg** header files, are given below. A discussion of the example is given in §5

Each `mesh[i]` is constructed with a log file named `mesh`i`.Log` and each has a dump file named `mesh`i`.Dump` showing the final `mesh[i]` incidence data. `mesh0.Log`, `mesh2.Log` and `mesh0.Dump` are shown in the subsequent subsection.

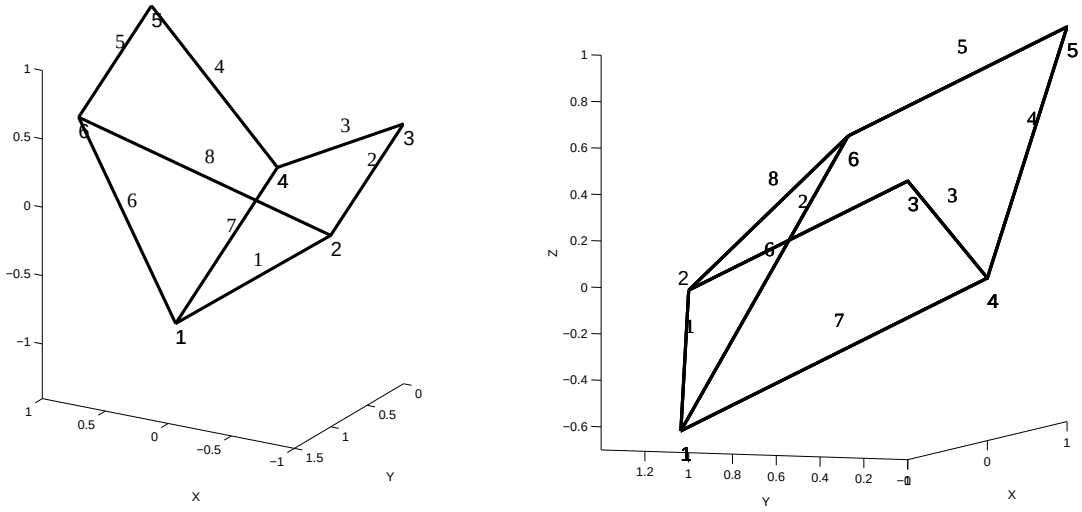


Figure 15: Boundary representation of simple trough example: 2 views

A.3.1 Tutorial example: simple mesh object header and implementation files

Project dependent class header files for tutorial example

```
class MVert : public BVert {

  public:
   double X,Y,Z;         // Euclidean 3-D coordinates
   MVert():BVert(){};    // default constructor
   MVert(int gVnum):BVert(gVnum){}; // semi explicit
```

```
   MVert(int vNum, double xc, double yc, double zc)
     { X = xc ; Y = yc ; Z = zc ; GVnum = vNum ; }; // explicit constructor
   ~MVert(){} ;
   int  CoordDiff(MVert *w) ;
          // returns 0 if this and w  deemed to have the same coordinates
   void Write();  //  outputs vertex data to stdout (extra member)
};

class MTri : public BTri {
  friend class Mesh2D ;

  public :
   static double MeshPlanePos[3] ; //a positive direction of mesh plane
            //  used temporarily to confirm orientation
        // of this MTri during Mesh2D::InsertTri

   MTri(): BTri() {};  // default constructor
   MTri(int gTnum); // explicit constructor
   ~MTri() ;                     // default destructor
   double Orientation(MVert** vbuf);
} ;


class MLSeg : public BLSeg {
  friend class Mesh2D ;

  public:
   MLSeg(): BLSeg() {};
   MLSeg(int gTnum): BLSeg(gTnum) {};
   ~MLSeg(){} ;
   double Orientation(MVert** vbuf);
};

#endif
```

Project dependent class implementations for tutorial example

```
#include "MVandT.h"
                        // the next three lines are used for CoordDiff
#include <math.h>       // provides fabs
#define macheps 1.0e-15  // floating point machine epsilon
#define max(a, b)  (a>b ? a : b)
```

```
int MVert::CoordDiff(MVert *w)
 //
{
double Xw = w->X ;
double Yw = w->Y ;
double Zw = w->Z ;
//
   if ( fabs(X-Xw) < 10.*macheps*max(fabs(X), fabs(Xw))
            &&
        fabs(Y-Yw) < 10.*macheps*max(fabs(Y), fabs(Yw))
            &&
        fabs(Z-Zw) < 10.*macheps*max(fabs(Z), fabs(Zw)) )
      {return 0;}
   return 1;
}

void MVert::Write(){
cout « " GVnum " « GVnum « " (" « X « ", " « Y ;
cout «  ", " « Z « ")" « endl;
};

  MTri::MTri(int gTnum ) { // explicit constructor
GTnum = gTnum ;
}


double MTri::Orientation(MVert ** vbuf)
 {
int egDummy;
double lEgDummy, inner;
double u[3],v[3],n[3] ;

   // get coords of  vbuf[1], vbuf[2] relative to vbuf[0]
        u[0] = vbuf[1]->X - vbuf[0]->X; v[0] = vbuf[2]->X - vbuf[0]->X;
        u[1] = vbuf[1]->Y - vbuf[0]->Y; v[1] = vbuf[2]->Y - vbuf[0]->Y;
        u[2] = vbuf[1]->Z - vbuf[0]->Z; v[2] = vbuf[2]->Z - vbuf[0]->Z;


//       compute n = normal to plane of triangle = vector cross product u x v

n[0] = u[1]*v[2]-u[2]*v[1] ;
n[1] = u[2]*v[0]-u[0]*v[2] ;
```

```
n[2] = u[0]*v[1]-u[1]*v[0] ;

inner = n[0]*MeshPlanePos[0] + n[1]*MeshPlanePos[1] +
                n[2]*MeshPlanePos[2] ;
return inner;
}

                 //  functions to satisfy virtual BTri::Orientation
double MLSeg::Orientation(MVert ** vbuf){return 0.0;}

//
```

A.3.2 Tutorial example: main program source code

```
//  tutorial example - simple trough

#include "MVandT.h"
#include "../Mesh2D.h"  // Note tricky relation of ifdefn MVandT
#include <stdio.h>
#include <stdlib.h>

int main()
{const int Nvtx = 6;
 double a, x[Nvtx], y[Nvtx], z[Nvtx] ;
 Mesh2D* mesh[3];
 MVert * vbuf[3];
 MTri  *nbuf[3];
 int cbuf[3], meshNo, i, vlab;

        // declare, and set up log files for, each mesh
 mesh[0] = new Mesh2D(1,"mesh0.Log");
 mesh[1] = new Mesh2D(1,"mesh1.Log");
 mesh[2] = new Mesh2D(1,"mesh2.Log");

 x[0] =  0 ; y[0] = 1.4 ; z[0] = -.7;
 x[1] = -1 ; y[1] = 1 ;   z[1] = 0 ;
 x[2] = -1 ; y[2] = 0 ;   z[2] = 0.5 ;
 x[3] =  0 ; y[3] = 0 ;   z[3] =  0 ;
 x[4] =  1 ; y[4] = 0 ;   z[4] = 1 ;
 x[5] =  1 ; y[5] = 1 ;   z[5] = 0.5 ;

       //  build boundary representation
 MVert* vstart = new MVert(1,x[0],y[0],z[0]);
 vbuf[0] = vstart;
 MVert* vdestn = new MVert(2,x[1],y[1],z[1]) ;
 vbuf[1] = vdestn;
 MLSeg* seg = new MLSeg(1);

 if(mesh[0]->InsertLSeg(seg,vbuf,nbuf,cbuf,mesh[2])==FAILED)
 {cout « " initial insert failed " « endl;
  return 0;
 };
     //  Now seg is registered in both mesh 0 and mesh 2
     //  and so are vstart and vdestn

 vlab = 3;    // explicit vertex label

     //  build open boundary at the top
 for( meshNo = 0; meshNo < 2 ; meshNo++)
 { for (i = 0; i<2 ; i++)
   { vbuf[0] = vdestn ;
     vdestn = new MVert(vlab,x[vlab-1],y[vlab-1],z[vlab-1]);
     vbuf[1] = vdestn;
     seg = new MLSeg(vlab-1);
     mesh[meshNo]->InsertLSeg(seg,vbuf,nbuf,cbuf,NULL);
     // these lsegs are joined only to outside.
     vlab++;
   };
 };
     // now all vertices are registered. Enter remaining
     // internal line segments for boundary rep
 vbuf[0] = vdestn;  vbuf[1] = vstart ;
 seg = new MLSeg(6);
 mesh[1]->InsertLSeg(seg,vbuf,nbuf,cbuf,mesh[2]) ;

 vbuf[0]=mesh[0]->VtxOfI(4); vbuf[1]=vstart;
 seg = new MLSeg(7);
 mesh[0]->InsertLSeg(seg,vbuf,nbuf,cbuf,mesh[1]) ;

 vbuf[0]=vdestn; vbuf[1] = mesh[0]->VtxOfI(2);
 seg = new MLSeg(8);
 mesh[2]->InsertLSeg(seg,vbuf,nbuf,cbuf,NULL) ;

           // boundary rep complete

    // add triangles in mesh[i], i = 0,1,2

 nbuf[0] = (MTri *) mesh[0]->LSegOfI(1) ; cbuf[0] = 0 ;
 nbuf[2] = (MTri *) mesh[0]->LSegOfI(7) ; cbuf[2] = 0 ;
 MTri * t = new MTri(1) ;
   // set MTri static member using +ve x axis as positive
   //  direction for mesh[0]
 t->MeshPlanePos[0] = 1.0 ;
 t->MeshPlanePos[1] = 0.0;  t->MeshPlanePos[2] = 0.0 ;

 mesh[0]->InsertTri(t,vbuf,nbuf,cbuf);

 nbuf[0] = (MTri *) mesh[0]->LSegOfI(2) ; cbuf[0] = 0 ;
 nbuf[1] = (MTri *) mesh[0]->LSegOfI(3) ; cbuf[0] = 0 ;
 nbuf[2] = t ; cbuf[2] = 1;
 t = new MTri(2) ;  //  t->MeshPlanePos already set

 mesh[0]->InsertTri(t,vbuf,nbuf,cbuf);

 nbuf[0] = (MTri *) mesh[1]->LSegOfI(6) ; cbuf[0] = 0 ;
 nbuf[1] = (MTri *) mesh[1]->LSegOfI(7) ; cbuf[1] = 1 ;
 t = new MTri(2) ;
   //  using -ve x axis as positive  direction for mesh[1]
 t->MeshPlanePos[0] = -1.0 ;
 t->MeshPlanePos[1] = 0.0;  t->MeshPlanePos[2] = 0.0 ;

 mesh[1]->InsertTri(t,vbuf,nbuf,cbuf);

 nbuf[0] = (MTri *) mesh[1]->LSegOfI(4) ; cbuf[0] = 0 ;
 nbuf[1] = (MTri *) mesh[1]->LSegOfI(5) ; cbuf[1] = 0 ;
 nbuf[2] = t ; cbuf[2] = 2;
 t = new MTri(4) ;

 mesh[1]->InsertTri(t,vbuf,nbuf,cbuf);

     //  finally - insert single triangle in mesh[2]
 nbuf[0] = (MTri*) mesh[0]->LSegOfI(1) ; cbuf[0]=1 ;
 nbuf[1] = (MTri*) mesh[1]->LSegOfI(6) ; cbuf[1]=1 ;
 nbuf[2] = (MTri*) mesh[2]->LSegOfI(8) ; cbuf[2]=0 ;
 t = new MTri(mesh[2]->NewGTnum()) ;
   //  using -ve y axis as positive  direction for mesh[2]
 t->MeshPlanePos[0] = 0.0 ;
 t->MeshPlanePos[1] = -1.0;  t->MeshPlanePos[2] = 0.0 ;

 mesh[2]->InsertTri(t,vbuf,nbuf,cbuf);

         // turn off recording  mesh transactions
 for(i=0; i<3 ; i++) mesh[i]->LogFileOnIsOne = 0;

 mesh[0]->Dump("mesh0.Dump","  mesh 0; tutorial ");
 mesh[1]->Dump("mesh1.Dump","  mesh 1; tutorial ");
 mesh[2]->Dump("mesh2.Dump","  mesh 2; tutorial ");


 return 1;
}
```

A.3.3 examples of debugging files

The `mesh0.Log` file

```
Mesh2D constructor: Vertices table size 50
                    Triangles table size 100
                    Boundary Reference table size 50
  InsertLSeg:  line segment 1    AddVtx: attempting to insert 1
    AddVtx: Successful
    AddVtx: attempting to insert 2
    AddVtx: Successful
   InsertLSeg: SUCCEEDED
  InsertLSeg:  line segment 2    AddVtx: attempting to insert 3
    AddVtx: Successful
   InsertLSeg: SUCCEEDED
  InsertLSeg:  line segment 3    AddVtx: attempting to insert 4
    AddVtx: Successful
   InsertLSeg: SUCCEEDED
  InsertLSeg:  line segment 7   InsertLSeg: SUCCEEDED
  InsertTri:  triangle 1
   InsertTri: SUCCEEDED
  InsertTri:  triangle 2
   InsertTri: SUCCEEDED
```

The `mesh2.Log` file

```
Mesh2D constructor: Vertices table size 50
                    Triangles table size 100
                    Boundary Reference table size 50
    AddVtx: attempting to insert 1
    AddVtx: Successful
    AddVtx: attempting to insert 2
    AddVtx: Successful
    AddVtx: attempting to insert 6
    AddVtx: Successful
  InsertLSeg:  line segment 8   InsertLSeg: SUCCEEDED
  InsertTri:  triangle 1
   InsertTri: SUCCEEDED
```

The `mesh0.Dump` file

Syntax for dump file listings:

triangle neighbours : [*GTnum* | -],<LSeg>
GTnum is label of the neighbour , - indicates `NULL`
optional flag `LSeg` indicates that the neighbour is an **MLSeg** .

vertex adjacencies : *GTnum*,<LSeg>,<other>
GTnum is label of the incident **MTri** or **MLSeg**
optional flag `LSeg` indicates the letter
optional flag `other` indicates that incident object is not registered in *this* mesh.

line segment neighbours : [*GTnum* | -],[`this` | `other`]
GTnum is label of the neighbouring **MTri** , - indicates `NULL` ,
second field designates *this* or other mesh.

```
  mesh 0; tutorial

Triangles:
  Number:1
  Vertices:   4, 1, 2
  Neighbours:  1,LSeg,  2,  7,LSeg,

  Number:2
  Vertices:   4, 2, 3
  Neighbours:  2,LSeg,  3,LSeg,  1,

Vertices:
  Number:1
  Adjacency: 1,other,  2,other,  1,  7,LSeg,  6,LSeg,other,  1,LSeg,

  Number:2
  Adjacency: 1,other,  2,  1,  8,LSeg,other,  2,LSeg,  1,LSeg,

  Number:3
  Adjacency: 2,  3,LSeg,  2,LSeg,

  Number:4
  Adjacency: 4,other,  2,other,  2,  1,  7,LSeg,  4,LSeg,other,  3,LSeg,

Line Segments:
  Number:1
  Vertices:   1, 2
  Neighbours:  1,this,  1,other,

  Number:2
  Vertices:   2, 3
  Neighbours:  2,this,  - ,
```

```
Number:3
Vertices:   3, 4
Neighbours:  2,this,  - ,

Number:7
Vertices:   4, 1
Neighbours:  1,this,  2,other,
```

References

[1] M Bern and D Eppstein. Mesh generation and optimal triangulation. In F K Huang, editor, *Computing in Euclidean Geometry*. World Scientific, 1992.

[2] P Breitkopf. Approche objet en elements finis. In *Revue europpeenne des elements finis*. Hermes, Paris, 1998.

[3] L P Chew. Guaranteed-quality mesh generation for curved surfaces. In *9th Annual Symposium on Comp Geometry*, pages 274–280, San Diego, California, 1993. ACM.

[4] M A Ellis and B Stroustrup. *The Annotated* `C++` *Reference Manual*. Addison-Wesley, 1990.

[5] A Fabri, G-J Geizeman, L Kettner, S Schirra, and S Schonherr. On the design of cgal, the computational geometry algorithms library. Technical Report 3407, INRIA, Sophia Antipolis, 1998.

[6] P L George. *Automatic Mesh Generation : application to finite element methods*. John Wiley and Sons, Paris : Masson, 1991.

[7] P L George and H Borouchaki. *Delaunay Triangulation and Meshing*. Hermes, 1998.

[8] L Guibas and J Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans on Graphics*, 4:74–123, 1985.

[9] L Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annual ACM Symposium, Comp. Geom.*, pages 146–154. ACM, 1998.

[10] R. Mackie. Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, 35:425–436, 1992.

[11] A V Mobley, M P Carroll, and S A Canann. An object oriented approach to geometry defeaturing for finite element meshing. In *Proc. 7th Intl Meshing Roundtable '98*. Sandia National Laboratories, 1998.

[12] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[13] R S Palmer. Chain models and finite element analysis: An executable chains formulation of plane stress. *Computer Aided Geometric Design*, 12:733–770, 1995.

[14] R B Simpson. A data base modeling abstraction for describing triangular mesh algorithms. *BIT*, 37:138–163, 1997.

[15] K Weihe. Using templates to improve C++ design. *C++ Report*, Feb:14–21, 1998.

Index

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399